Handout Progress

Interpretation and Compilation
15-NOV-2020

Luis Caires

Handout Phase 0.1

Implement a complete interpreter and compiler for a tiny arithmetic expression language

Use the approach we are developing in the course

- LL(1) parser using JAVACC
- AST Model
- Interpreter
- Compiler

Fully understanding the handout statement is part of the handout as well.

Contact me anytime if you need help.

Handout Phase 0.1

Learning Outcomes

- you learn how to develop a simple parser using JavaCC
  - understand how to specify tokens using regular expressions
  - you understand how to specify a simple non ambiguous LL(1) context free grammar
- you understand the basics of abstract syntax trees (AST)
- you learn how to define the semantics evaluation function over the AST (this provides an interpreter for the language)
- you learn how to define the semantics compilation function over the AST (this provides a compiler for the language, and allows you to meet the Java Virtual Machine (JVM) internals

Handout Phase 0.1

Abstract Syntax (Abstract Constructors)

ADD: Exp x Exp -> Exp
SUB: Exp x Exp -> Exp
MUL: Exp x Exp -> Exp
DIV: Exp x Exp -> Exp
UMINUS: Exp -> Exp
NUM: int -> Exp

Handout Phase 0.1

Concrete Syntax (Examples)

2*3+4

2*(3+4)

4-2/5*2

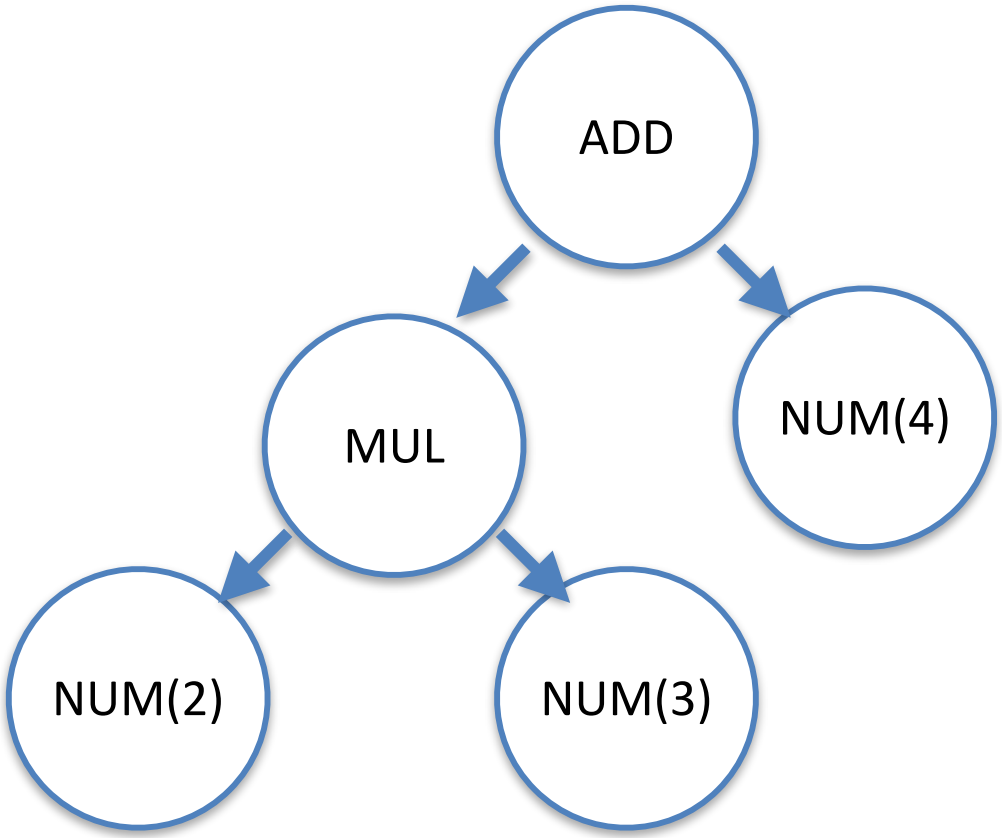
-(2+2-4)

-2

Handout Phase 0.1

Abstract Syntax (Abstract Constructors) **2*3+4**

ADD(MUL(NUM(2),NUM(3)), NUM(4))

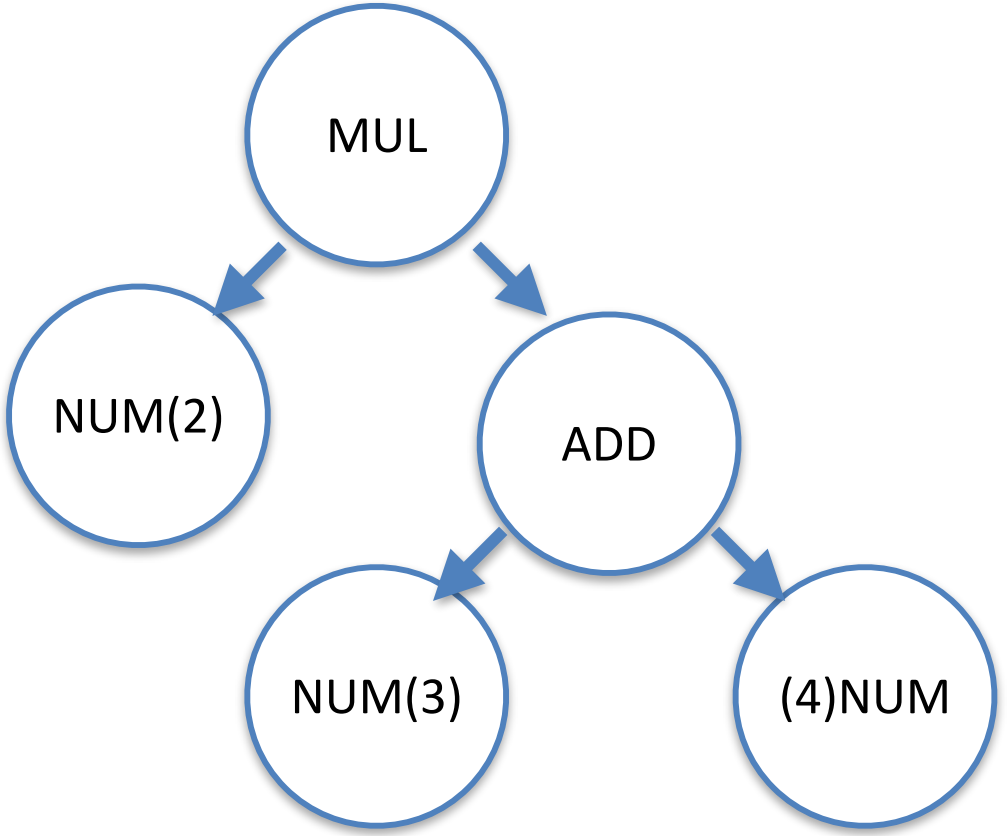


Handout Phase 0.1

Abstract Syntax (Abstract Constructors)

2*(3+4)

MUL(NUM(2), ADD(NUM(3),NUM(4))



Handout Phase 0.1

Grammar

Alphabet = { **num**, **+**, **-**, *, **/**, **(**, **)** }

E -> **num**
E -> E **+** E
E -> E **-** E
E -> E ***** E
E -> E **/** E
E -> **-** E
E -> **(** E **)**

Handout Phase 0.1

Grammar (ambiguous)

E ->

| **num**

| E **+** E | E **-** E

| E ***** E | E **/** E | **-**E | **(** E **)**

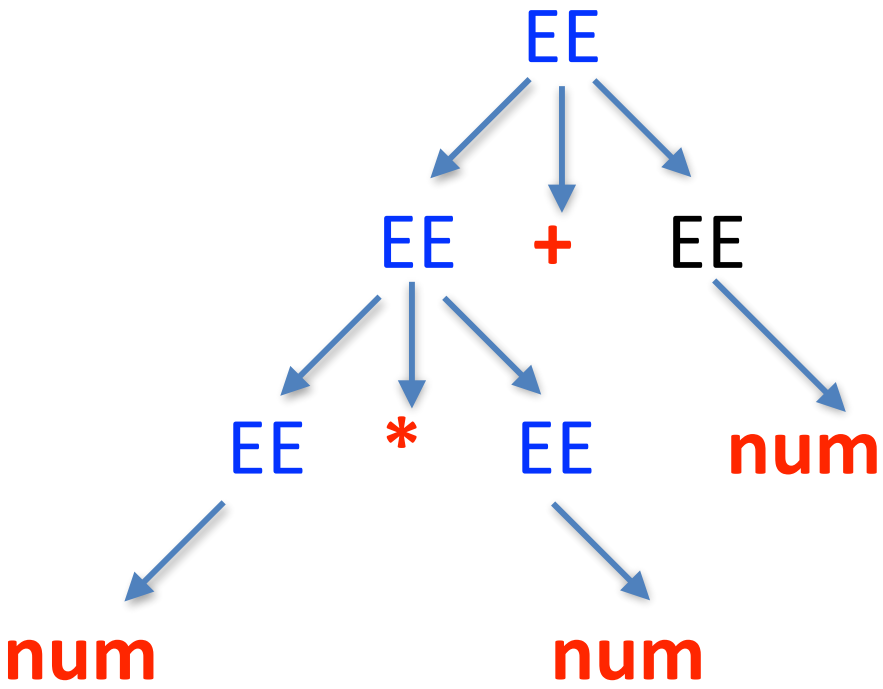
num * num + num has two derivations

EE -> EE **+** EE -> EE ***** EE **+** EE -> **num * num + num**

EE -> EE ***** EE -> EE ***** EE **+** EE -> **num * num + num**

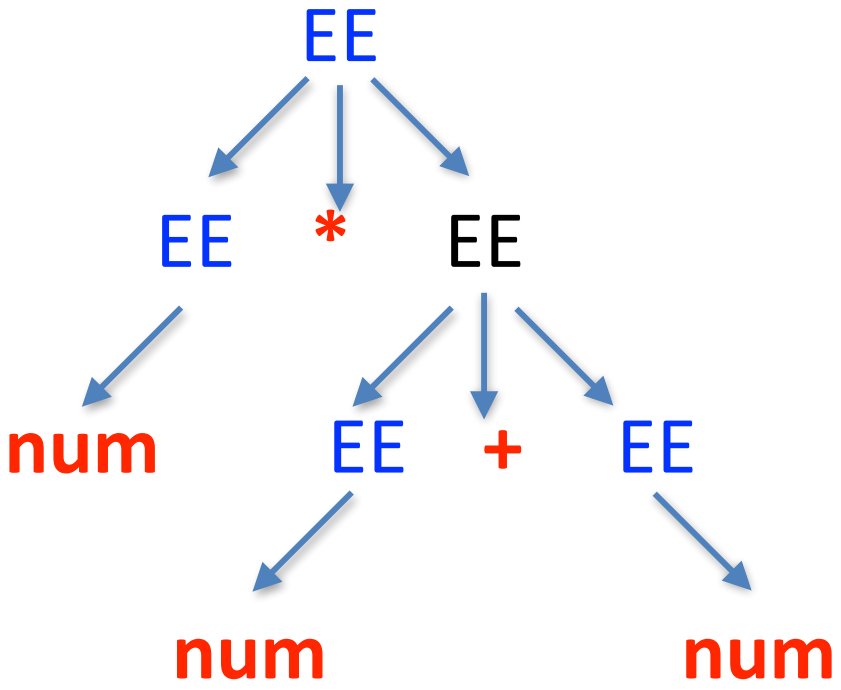
Handout Phase 0.1

EE -> EE + EE -> EE * EE + EE -> num * num + num



Handout Phase 0.1

EE -> EE * EE -> EE * EE + EE -> num * num + num



Handout Phase 0.1

Grammar (non-ambiguous LL(1))

E -> T

E -> T + E

T -> F

T -> F * T

F -> num

F -> (E)

F -> - F

Handout Phase 0.1

Grammar (non-ambiguous)

E -> T

E -> T + E

T -> F

T -> F * T

F -> num

F -> (E)

F -> - F

Handout Phase 0.1

Grammar (non-ambiguous and LL(1))

E -> TE’

E’ -> ε | **+** E

T -> FT’

T’ -> ε | ***** T

F -> **num**

F -> **(** E **)**

F -> **-** F

E -> TE’ -> T+E -> T+TE’ -> T +T+E -> … -> T + T + … + T

Handout Phase 0.1

Grammar (non-ambiguous and LL(1))

E -> TE’

E’ -> ε | **+** E

T -> FT’

T’ -> ε | ***** T

F -> **num**

F -> **(** E **)**

F -> **-** F

num * num + num

E -> TE’ -> FT’E’ -> **num** T’E’ ->

num * T E’ -> **num *** FT’ E’ ->

num * num T’ E’ ->

num * num + E ->

num * num + E ->

num * num + E ->

num * num + TE’ -> num * num + num

E -> TE’ -> T+E -> T+TE’ -> T +T+E -> … -> T + T + … + T

Handout Phase 0.1

Grammar (non-ambiguous and LL(1))

EBNF (Extended BNF)

E -> T [(**+** | **-**) T] *
T -> F [(***** | **/**) F] *
F -> **num** | **(** E **)** | **-** **F**

AST (schematic)

```
interface ASTNode {
int eval() ...
}

class AST??? implements ASTNode {

}
```

AST (schematic)

```
class ASTAdd implements ASTNode {
ASTNode lhs;
ASTNode rhs;
public ASTAdd (ASTNode l, ASTNode r) {
        lhs = l;
        rhs = r;
}
}
```

AST (schematic)

```
class ASTAdd implements ASTNode {


public eval() {
        int vl = lhs.eval();
        int rv = rhs.eval();
        return vl + rv;
}


}
```

interpreter main (schematic) (schematic)

```
PARSER_BEGIN(Parser0)

public class Parser0 {

  /** Main entry point. */
  public static void main(String args[]) {
    Parser0 parser = new Parser0(System.in);

    while (true) {
    try {
    System.out.print( "> " );
    ASTNode ast = parser.Start();
    System.out.println( ast.eval() );
    } catch (Exception e) {
      System.out.println ("Syntax Error!");
      parser.ReInit(System.in);
    }
    }
  }

}

PARSER_END(Parser0)
```

Handout Phase 0.2

Learning Outcomes

- you learn how to develop a simple parser using JavaCC
  - understand how to specify tokens using regular expressions
  - you understand how to specify a simple non-ambiguous LL(1) context free grammar
- you understand the basics of abstract syntax trees (AST)
- you learn how to define the semantics evaluation function over the AST (this provides an interpreter for the language)
- you learn how to define the **semantics compilation** function over the AST (this provides a compiler for the language, and allows you to meet the **Java Virtual Machine** (JVM) internals

AST (schematic)

```
interface ASTNode {
int eval();
void compile(CodeBlock c);
}

class CodeBlock {
String code[];
int pc;
void emit(String opcode){
        code[pc++] = opcode;
}
void dump(PrintStream f) { … // dumps code to f }
}
```

AST (schematic)

```
class ASTAdd implements ASTNode {

public void compile(CodeBlock c) {
        lhs.compile(c);
        rhs.compile(c);
        c.emit(“iadd”);
}

}
```

JVM bytecodes

- sipush *n*
- iadd
- imul
- isub
- idiv
- …

```
.class public Main
.super java/lang/Object
;
; standard initializer
.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method
.method public static main([Ljava/lang/String;)V
    .limit locals 10
    .limit stack 256
    ;   1 - the PrintStream object held in java.lang.System.out
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ; place your bytecodes here between START and END
    ; START

    sipush 20
    sipush 20
    iadd
    sipush 2
    imul

    ; END
    ; convert to String;
    invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
    ; call println
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    return
.end method
```

JVM bytecodes

sipush

Operation

Push `short`

Format

```
sipush
byte1
byte2
```

Forms

sipush = 17 (0x11)

Operand Stack

... →

..., *value*

Description

The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate `short`, where the value of the `short` is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

JVM bytecodes

iadd

Operation

Add `int`

Format

`iadd`

Forms

iadd = 96 (0x60)

Operand Stack

..., *value1*, *value2* →

..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

compiler main (schematic)

```
PARSER_BEGIN(Parser0C)

  public static void main(String args[]) {
    Parser0C parser = new Parser0C(System.in);
    CodeBlock code = new CodeBlock();
    while (true) {
    try {
    ASTNode ast = parser.Start();
    ast.compile(code);
    code.dump(outfile);
    } catch (Exception e) {
      System.out.println ("Syntax Error!");
      parser.ReInit(System.in);
    }
    }
  }

PARSER_END(Parser0C)
```

Summary of what to know / do

- Install and run javacc
- Write javacc grammars for expression languages
- Define in Java the AST for expression language
- Implement an interpreter method (eval) in the AST for evaluating expressions in read-eval-print loop
- Understand the basic internals of Java Virtual Machine, and the instructions needed to compile expressions
- Install and run jasmin (JVM Assembler)
- Implement the compiler method (compile) in the AST for translating expressions to JVM code and generates JVM code using jasmin (use Main.j as stub).

Summary of what to know / do

- Remove the main() method from Parser0.jj and place it in two different “main” classes
    - ICLInterpreter
    - ICLCompiler
- The interpreter runs a read-eval-print loop as before

  > ICLInterpreter

  > 2+3

  5

  > 4/2

  2

Summary of what to know / do

- Remove the main() method from Parser0.jj and place it in two different “main” classes
    - ICLInterpreter
    - ICLCompiler
- Make your compiler accept a source file name in the command line and execute jasmin on the generated assembler file, so that it actually generates the output class file directly.
- So, If text file source.icl contains just the line 2+3, then

  > ICLCompiler source.icl

  > java source

  5

compiler main (schematic)

```
public class ICLCompiler {

  public static void main(String args[]) {
    Parser parser = new Parser(System.in);
    CodeBlock code = new CodeBlock();
    while (true) {
    try {
    ASTNode ast = parser.Start();
    ast.compile(code);
    code.dump(outfile);
    } catch (Exception e) {
      System.out.println ("Syntax Error!");
      parser.ReInit(System.in);
    }
    }
  }
```

Handout Phase 2

Interpretation and Compilation
25-OUT–2020

Luis Caires

What to do

Implement an interpreter for expression language with definitions

Use the approach developed in the lectures

- Extend your JAVACC LL(1) parser
  - Extend your parser with ids and definitions
- Extend your AST Model
  - ASTId, ASTDef
  - Add actions to the parser so that it will build an AST for correct input expressions
- Define the interpreter (eval method)
- You will need to define an environment based semantics

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

CALC Interpreter (environment based)

- **Algorithm eval( ) that computes the denotation (integer value) of any open CALCI expression:**

$$\mathbf{eval} : \text{CALCI} \times \text{ENV} \rightarrow \text{Integer}$$

eval(**num**(*n*) , *env*) ≜ *n*

eval(**id**(s) , *env*) ≜ *env*.Find(s)

eval(**add**(E1,E2) , *env*) ≜ **eval**(E1, *env*) + **eval**(E2, *env*)

…

```
eval( def(s, E1, E2), env) ≜ [ v1 = eval(E1, env);
                               env = env.BeginScope();
                               env =  env.Assoc(s, v1);
                               val = eval(E2, env);
                               env = env.EndScope();
                               return val ]
```

- **Note: Case of id(s) implemented by lookup of the value of s in the current environment**

Language with definitions

Abstract Syntax

EE ->

| **num** | **id**

| EE **+** EE | EE **-** EE

| EE ***** EE | EE **/** EE | **-**EE | **(** EE **)**

| **def** (**id** = EE)+ **in** EE **end**

Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;

def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;

def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```

AST

```
interface ASTNode {
int eval(Environment e);
}


class AST??? implements ASTNode {

}
```

AST

```
public class ASTId implements ASTNode {
String id;
ASTId(…) {…}
int eval(Environment e) {
    return e.find(id);
}
}

class AST??? implements ASTNode {


}
```

AST

```
public class ASTMul implements ASTNode {
ASTNode lhs, rhs;
ASTMul(…) {…}
int eval(Environment e) {
    return lhs.eval(e) * rhs.eval(e);
}
}

class AST??? implements ASTNode {


}
```

AST

```
class ASTDef implements ASTNode {
List<Pair<String,ASTNode>> init;
ASTNode body;
int eval(Environment e) {
    ….
}
}
```

AST

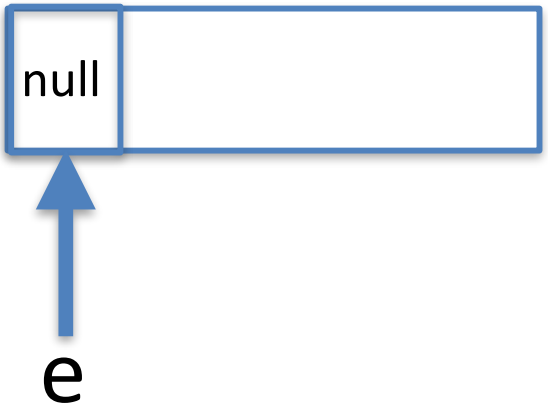
```
class ASTDef implements ASTNode {
Map<String,ASTNode> init;
ASTNode body;
int eval(Environment e) {
    ….
}
}
```

Environment (interpreter)

```
class Environment {
Environment beginScope(); //— push level
Environment endScope(); // - pop top level
void assoc(String id, int val);
int find(String id);
}
```

Environment (interpreter)

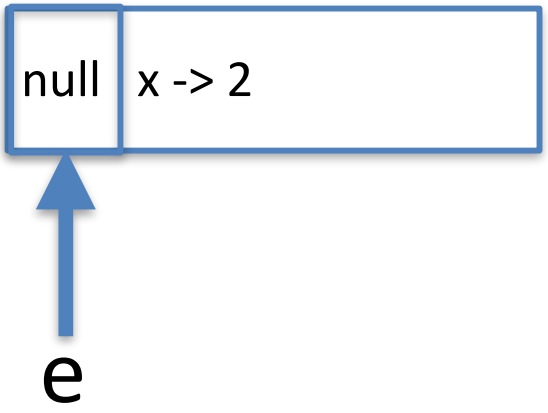
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```



e = new Environment()

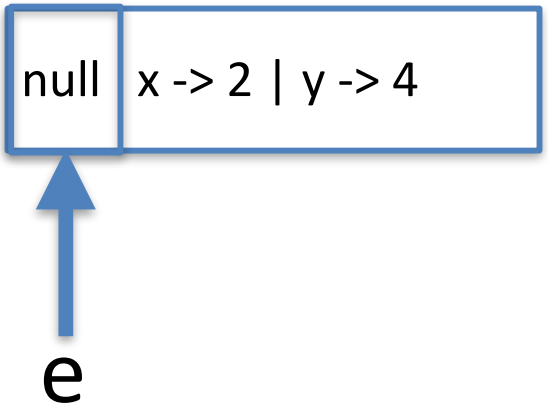
Environment (interpreter)

```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}

e.assoc("x",2)
```



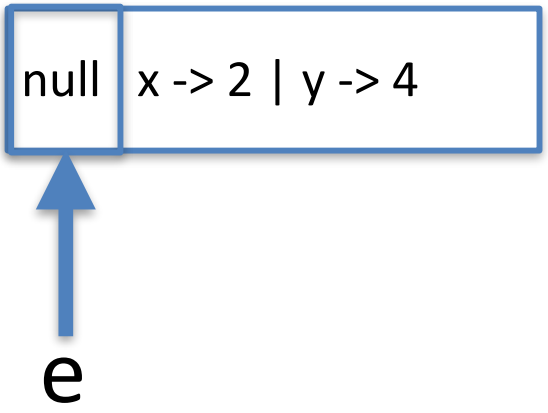
Environment (interpreter)

```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}

e.assoc("y",4)
```



Environment (interpreter)

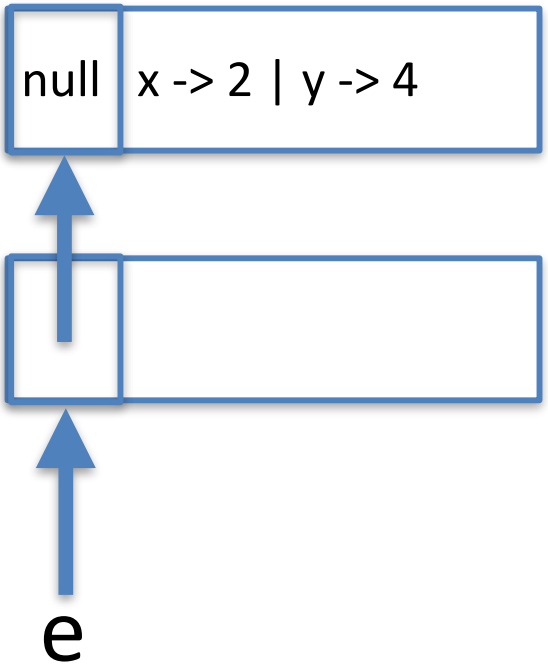
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```



e.assoc(“y”,5) raise exception IDDeclaredTwice

Environment (interpreter)

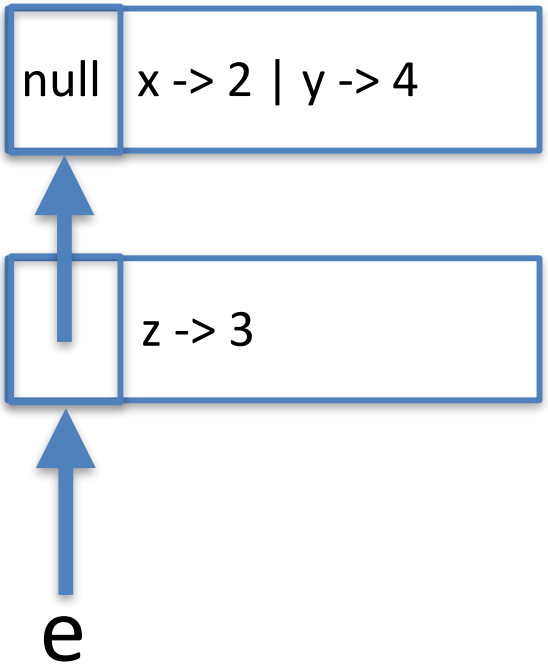
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```




```
e = e.beginScope();
```

Environment (interpreter)

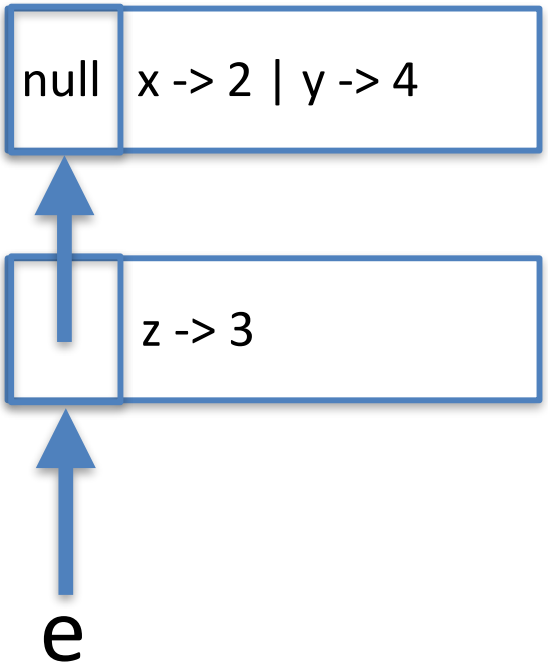
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```



e.assoc(“z”,3)

Environment (interpreter)

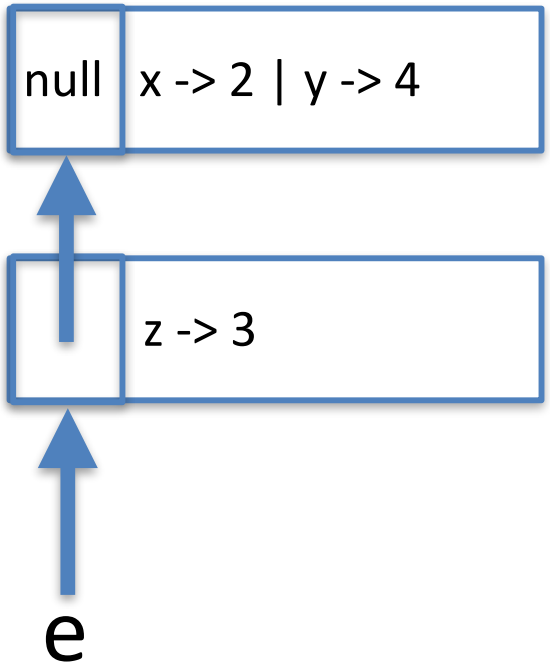
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```



e.find(“z”) returns 3

Environment (interpreter)

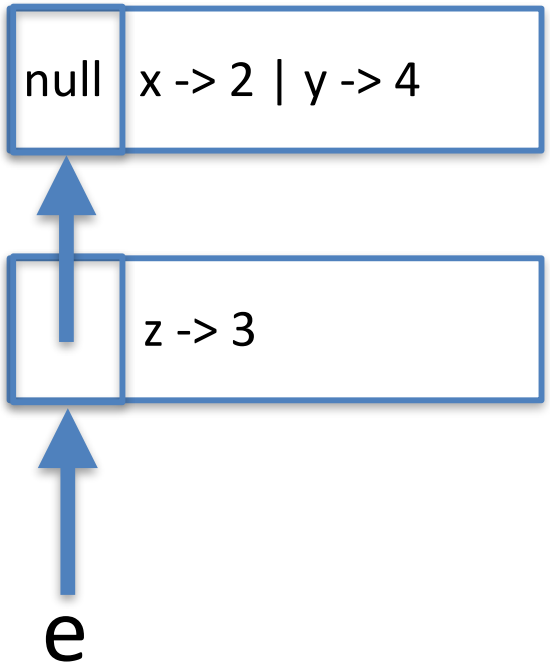
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```



e.find(“x”) returns 2

Environment (interpreter)

```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```

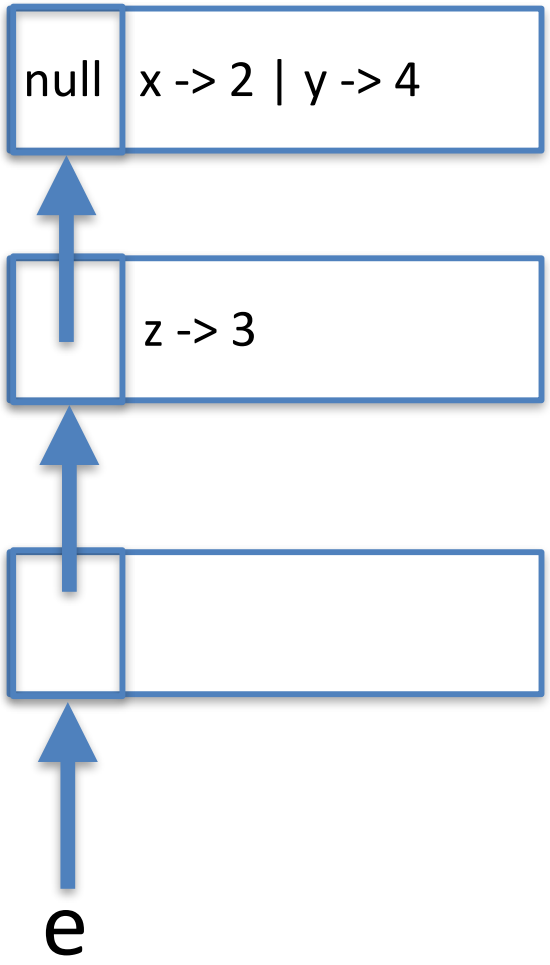


e.find(“a”) raises “Undeclared Identifier”

Environment (interpreter)

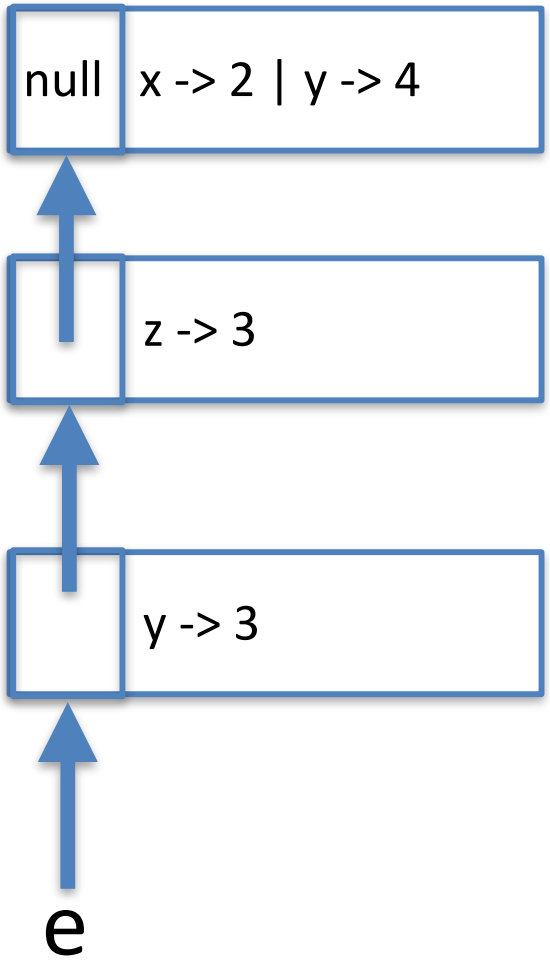
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```

```
e = e.beginScope();
```



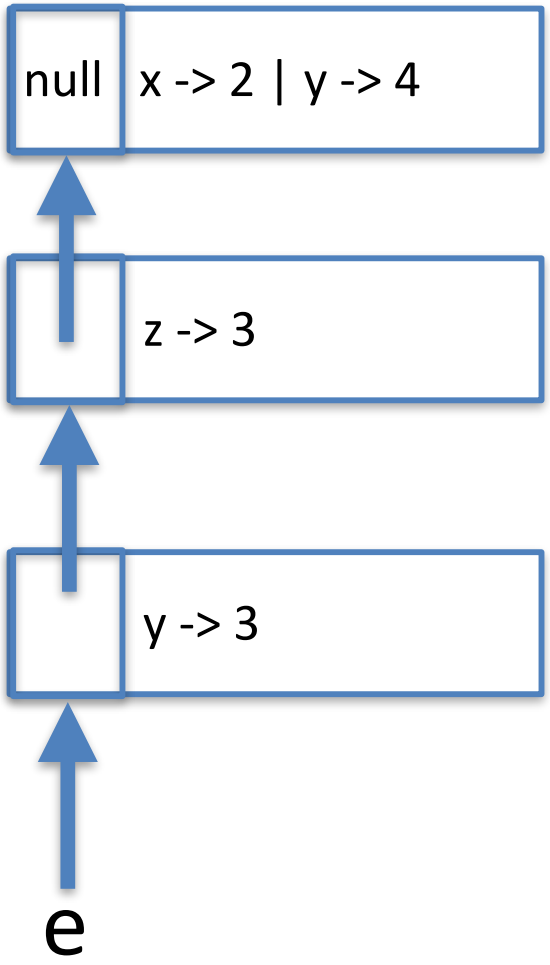
Environment (interpreter)

```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}


e.assoc("y",3);
```



Environment (interpreter)

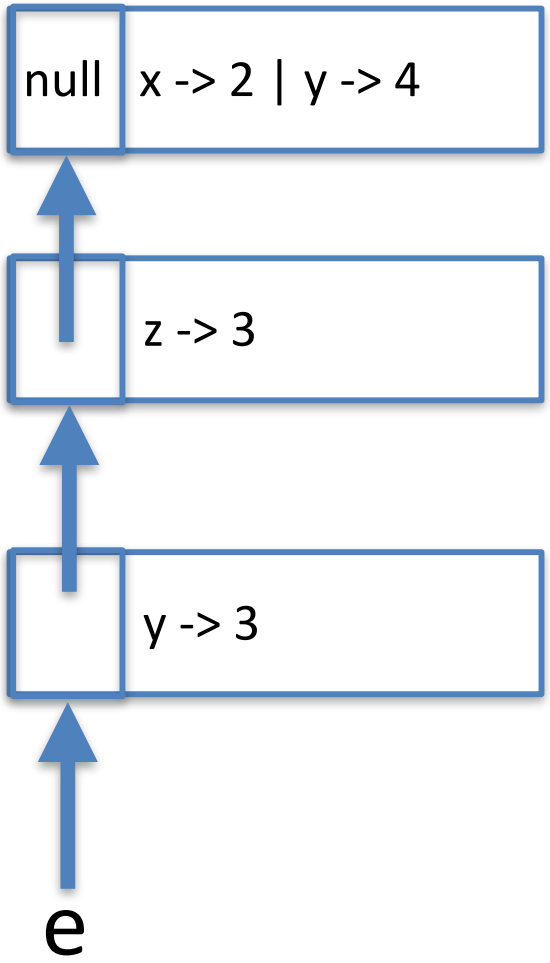
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}


e.assoc("y",3);
```



Environment (interpreter)

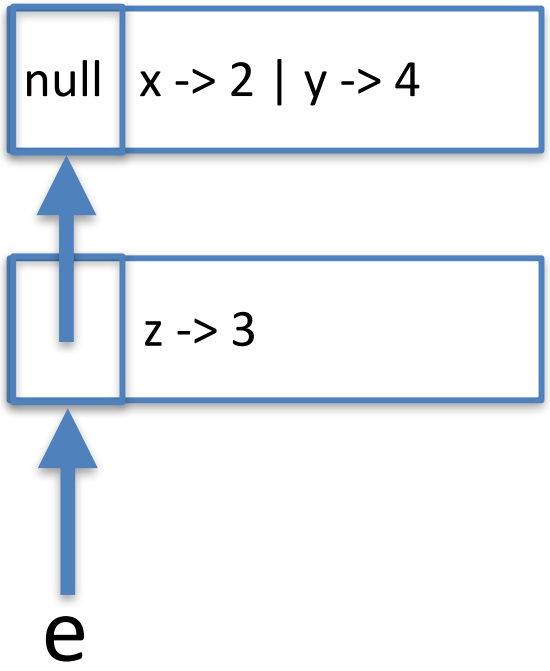
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```

e.find(“y”) returns 3



Environment (interpreter)

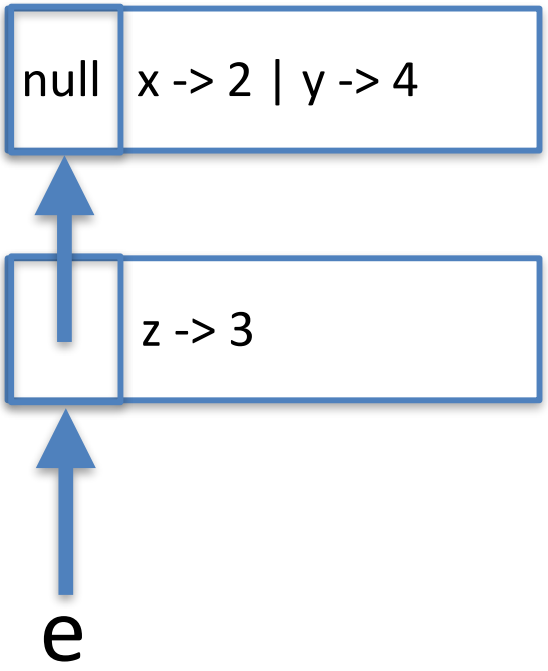
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```




```
e.endScope()
```

Environment (interpreter)

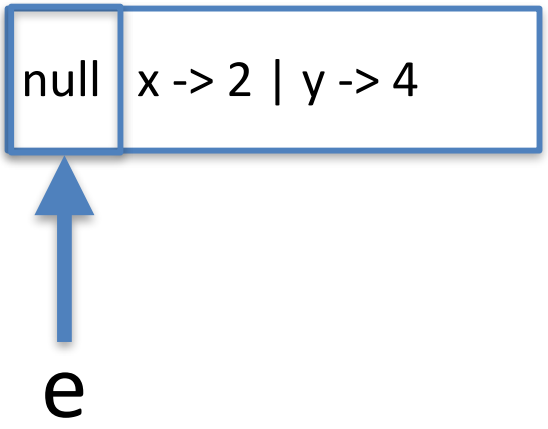
```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```



e.find(“y”) returns 4

Environment (interpreter)

```
class Environment {
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}

e.endScope()
```



Environment (interpreter)

```
class Environment {
Environment ancestor;
Environment beginScope();
Environment endScope();
void assoc(String id, int val);
int find(String id);
}
```

e == null

e = e.endScope()

Sample programs

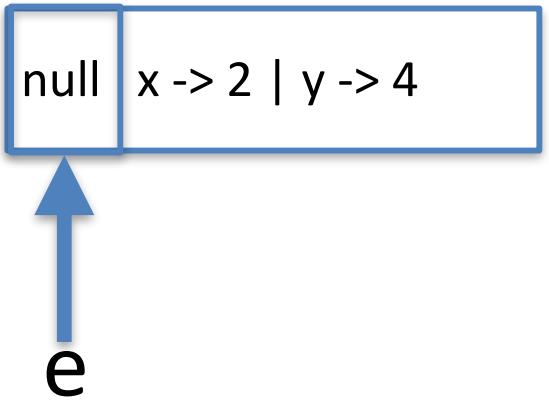
```
def x = 1 in
  def y = x+x in x + y end end;;

def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;

def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```

Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;
```

```
def x = 2
    y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```

```
def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```

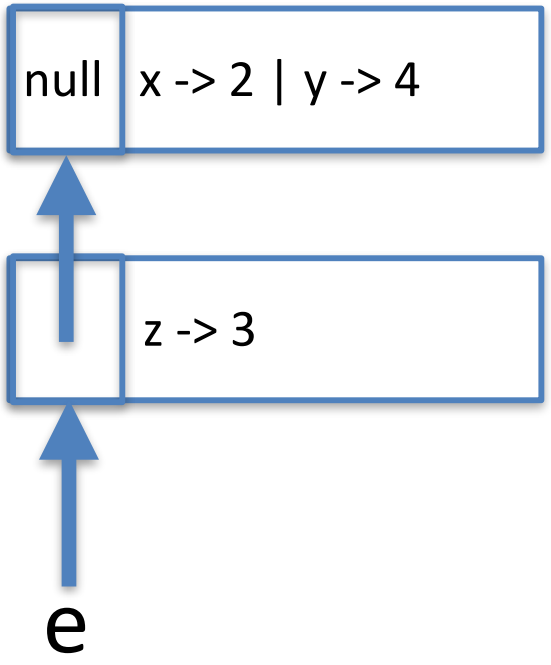


Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;
```

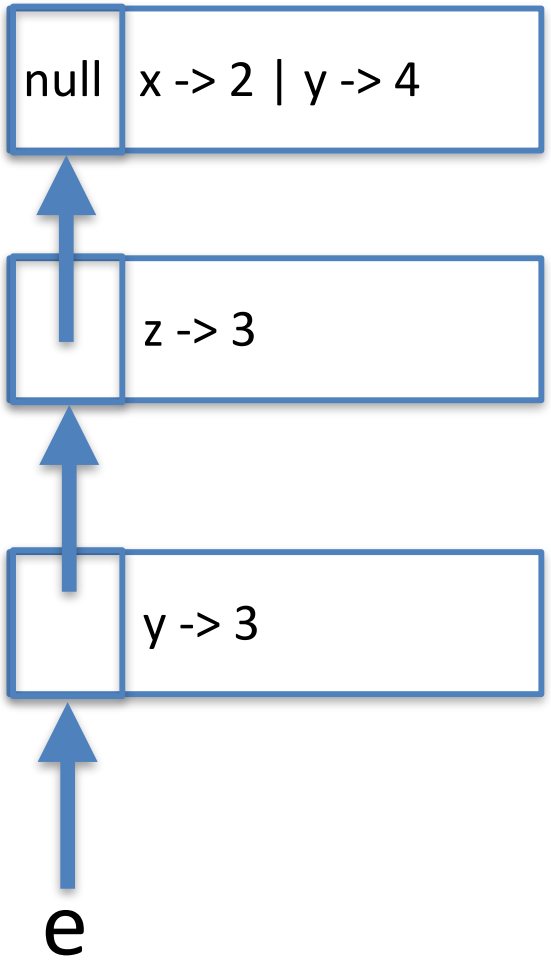
```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```

```
def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```



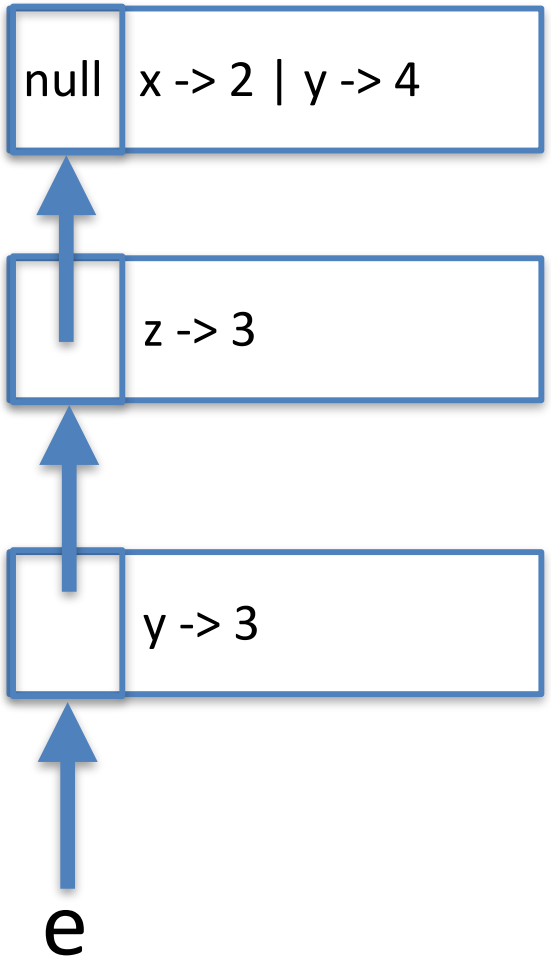
Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;

def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;

def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```



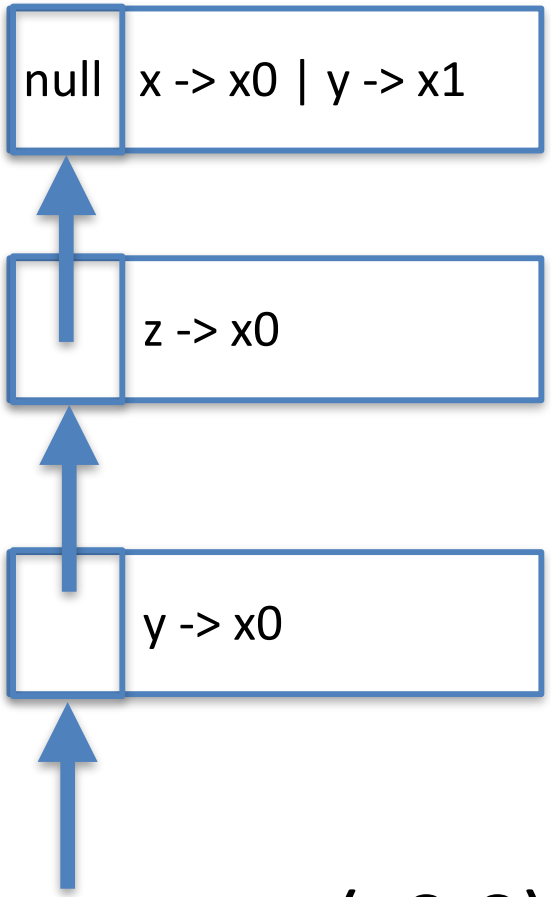
Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;

def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
   x + y + z end end end;;

def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```



Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;

def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
   x + y + z end end end;;

def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```

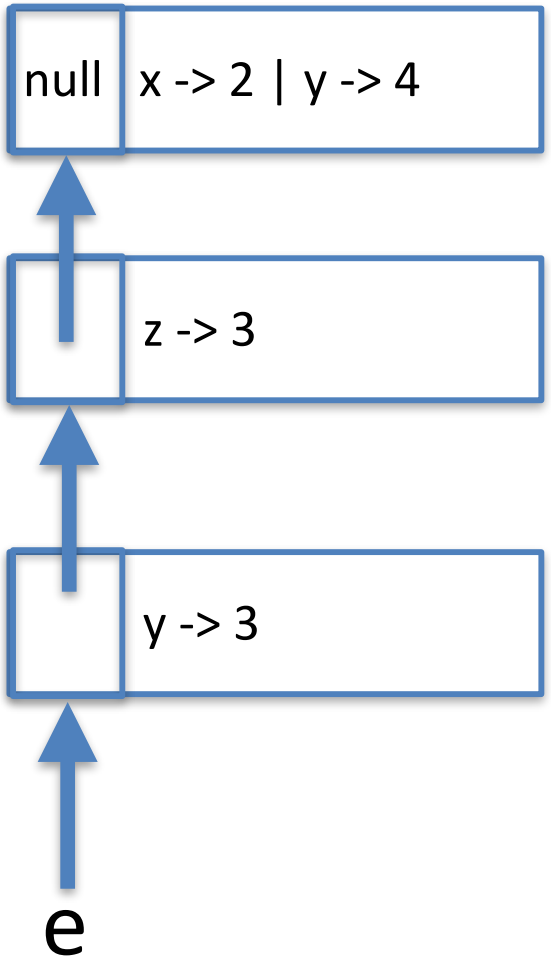


x -> (x0,0)

z -> (x0.1)

Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;

def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;

def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```

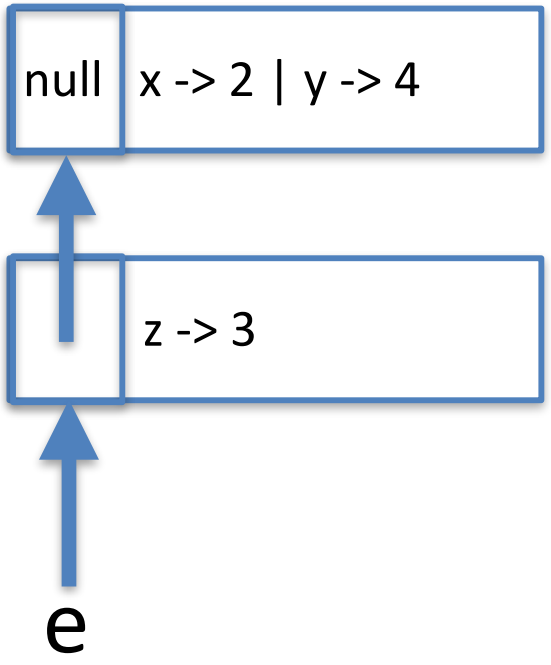


Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;
```

```
def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```

```
def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```

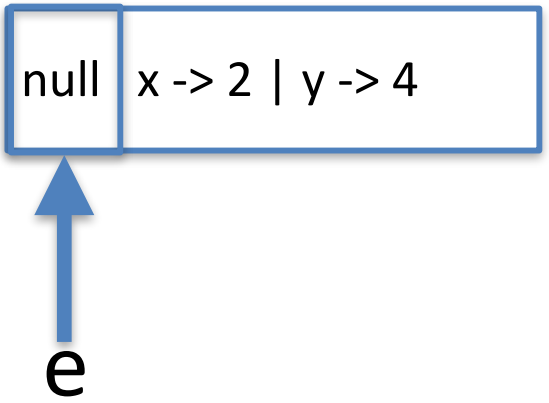


Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;
```

```
def x = 2
    y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;
```

```
def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```



Sample programs

```
def x = 1 in
  def y = x+x in x + y end end;;

def x = 2
      y = x+2 in
def z = 3 in
  def y = x+1 in
    x + y + z end end end;;

def x = 2 in
  def y = def x = x+1 in x+x end
  in x * y end end;;
```

Scheme

E -> T (+ T)*

T -> F (* F) *

F -> <num>
| <id>
| def { l = new List }
(id = E { l.add (..) }) + in b = E end
{ return new ASTDef(l, b) }

Scheme

```
def
  x = 1
  y = x + x
in
    def  z = x + y in z + z end
    +
    def  k = x * 2 in k + k end
end
```

What to do

Implement an compiler for expression language with definitions

Use the approach developed in the lectures

- Extend your JAVACC LL(1) parser
  - Extend your parser with ids and definitions
- Extend your AST Model
  - ASTId, ASTDef
  - Add actions to the parser so that it will build an AST for correct input expressions
- Define the compiler (compile method) - see Recitation slides block 4A
- You will need to define a compiler environment, assigning coordinates to identifiers.

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

AST

interface ASTNode {
int eval(Environment e);
void compile(CodeBlock c; *[Environment e]*);
}

class AST??? implements ASTNode {

}

CodeBlock (naive sketch)

```
class CodeBlock {
String code[];
int pos;

void emit(String bytecode) {
   code[pos] = bytecode;
   pos ++;
}

String gensym() { … }

void dump(PrintStream f) {
…
}

}
```

AST

```
class ASTAdd implements ASTNode {
ASTNode lhs, rhs;
…
void compile(CodeBlock c, Environment env) {
        lhs.compile(c, env);
        rhs.compile(c, env);
        c.emit(“iadd”);
}
```

AST

```
class ASTDef implements ASTNode {
String id;
ASTNode init;
ASTNode body;
…
void compile(CodeBlock c, Environment env) {

}
```

AST

```
class ASTDef implements ASTNode {
List<Bind> bindings; // each Bind a pair (String, ASTNode)
ASTNode body;
…
void compile(CodeBlock c, Environment env) {

}
```

Environment (compiler)

```
class Environment {
Environment beginScope(); //— push level
Environment endScope(); // - pop top level
int depth(); // - returns depth of “stack”
void assoc(String id, Coordinates c);
Coordinates find(String id);
}
env.find(“x”) -> (1,”x2”)
level-shift(“x”) =env.depth()-1
```

Environment (generic)

```
class Environment<X> {
Environment beginScope(); //— push level
Environment endScope(); // - pop top level
int depth(); // - returns depth of “stack”
void assoc(String id, X bind);
X find(String id);
}
```

Compilation of def blocks (example)

```
def
   x  = 2  |-> (0,”v0”)
   y  = 3  |-> (0,”v1”)
in
  def
     k = x + y   // for y = level_shift(“y”) = 1
  in
     x + y + k  // for y = level_shift(“y”) = 2
  end
end;;
```

AST

```
class ASTDef implements ASTNode {

void compile(CodeBlock c, Environment env) {
      // def x1 = E1 … xn = En in Body end
       env = env.beginScope();
      // generate code for frame init and link into RT env
       // for each x = E
                  c.emit(“aload 3”);
                  E.compile(c,env) ;
                  c.emit(“putfield “+frame+ “…./s1”);
                  env = env.assoc(x1, (env.depth, “s1”));
       //  Body.compile(c, env);
      // generate code for frame pop off
    env.endScope();
}
```

Compilation of def blocks (example)

```
.class public frame_0
.super java/lang/Object
.field public sl Ljava/lang/Object;
.field public v0 I
.field public v1 I

.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return


.end method
```

default constructor (JVM requires it)

```
.class public frame_1
.super java/lang/Object
.field public sl Lframe_0;
.field public v0 I

.end method
```

```
def
  x  = 2
  y  = 3
in
 def
     k = x + y
 in
     x + y + k
 end
end;;
```

Compilation of def blocks (example)

```
new frame_0
dup
invokespecial frame_0/<init>()V
dup
aload_3
putfield frame_0/sl Ljava/lang/Object;
astore_3
aload_3
sipush 2
putfield frame_0/v0 I
aload_3
sipush 3
putfield frame_0/v1 I
new frame_1
dup
invokespecial frame_1/<init>()V
dup
aload_3
putfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
putfield frame_1/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v0 I
aload_3
getfield frame_1/sl Lframe_0;
getfield frame_0/v1 I
iadd
aload_3
getfield frame_1/v0 I
iadd
aload_3
getfield frame_1/sl Lframe_0;
astore_3
aload_3
getfield frame_0/sl Ljava/lang/Object;
astore_3
```

```
def
  x  = 2
  y  = 3
in
  def k = x + y
  in
       x + y + k
  end
end;;
```

JVM bytecodes

JVM bytecodes

sipush

Operation

Push `short`

Format

```
sipush
byte1
byte2
```

Forms

sipush = 17 (0x11)

Operand Stack

... →

..., *value*

Description

The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate `short`, where the value of the `short` is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

JVM bytecodes

iadd

Operation

Add `int`

Format

`iadd`

Forms

iadd = 96 (0x60)

Operand Stack

..., *value1*, *value2* →

..., *result*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

JVM bytecodes

dup

Operation

Duplicate the top operand stack value

Format

dup

Forms

dup = 89 (0x59)

Operand Stack

..., *value* →

..., *value*, *value*

Description

Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

The *dup* instruction must not be used unless *value* is a value of a category 1 computational type (§2.11.1).

JVM bytecodes

aload

Operation

Load `reference` from local variable

Format

```
aload
index
```

Forms

aload = 25 (0x19)

Operand Stack

... →

..., *objectref*

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes

The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction (§*astore*) is intentional.

The *aload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

JVM bytecodes

astore

Operation

Store `reference` into local variable

Format

```
astore
index
```

Forms

astore = 58 (0x3a)

Operand Stack

..., *objectref* →

...

Description

The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

Notes

The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clause of the Java programming language (§3.13).

The *aload* instruction (§*aload*) cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *astore* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

JVM bytecodes

new

Operation

Create new object

Format

```
new
indexbyte1
indexbyte2
```

Forms

new = 187 (0xbb)

Operand Stack

... →

..., *objectref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in a class type. Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (§2.3, §2.4). The *objectref*, a `reference` to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

JVM bytecodes

putfield

Operation

Set field in object

Format

```
putfield
indexbyte1
indexbyte2
```

Forms

putfield = 181 (0xb5)

Operand Stack

..., *objectref*, *value* →

...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The class of *objectref* must not be an array. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

The referenced field is resolved (§5.4.3.2). The type of a *value* stored by a *putfield* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class, and the instruction must occur in an instance initialization method (`<init>`) of the current class (§2.9).

The *value* and *objectref* are popped from the operand stack. The *objectref* must be of type `reference`. The *value* undergoes value set conversion (§2.8.3), resulting in *value*', and the referenced field in *objectref* is set to *value*'.

JVM bytecodes

getfield

Operation

Fetch field from object

Format

```
getfield
indexbyte1
indexbyte2
```

Forms

getfield = 180 (0xb4)

Operand Stack

..., *objectref* →

..., *value*

Description

The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2). The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.

The type of *objectref* must not be an array type. If the field is `protected`, and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class.

Multiple (basic) Data Types

Part 1: interpreter and dynamic typechecking

Interpretation and Compilation
15-NOV-2020

Luis Caires

Basic integer and boolean operations

Arithmetic operations (on integer values)

E+E, E-E, E*E, E/E, -E

Relational operations

E==E, E>E, E<E, E<=E, E>=E

Logical operations (on boolean values)

E && E, E || E, ~E

Language with Basic Data Types

Abstract Syntax

EE ->

| **num** | **true** | **false** | **id**
| EE **+** EE | EE **-** EE
| EE ***** EE | EE **/** EE | **-**EE | **(** EE **)**
| EE **==** EE | EE **>** EE | EE **>=** EE | **…**
| EE **&&** EE | EE **||** EE | **~** EE
| **def** (**id** = EE)+ **in** EE **end**

Parsing (concrete syntax)

```
BA -> BM (|| BM)*          // boolean additives
BM -> R (&& R)*            // boolean multiplicatives
R -> E ( > E )?            // relops
E -> T ( + T)*             // integer additives
T -> F ( * F) *            // integer multiplicatives
F -> num
   | id  | true | false | ( BA )
   | - F  {...} | ~ F
   | def  { l = new List }
            ( id = BA() { l.add (.. ) }  ) + in  b = BA() end
    { return new ASTDef( l, b ) }
```

IValue (generic)

```
interface IValue { /* represents values */
String show();
}

IValue eval(Environment<IValue> env) { ... }

VInt(n)
VBool(t)
…
```

IValues (schematic)

```
class VInt implements IValue {
int v;
VInt(int v0) { v = v0; }
int getval() { return v;}
}
```

Interpreter with Dynamic Type Checking (idea)

```
class ASTAdd implements ASTNode {

IValue eval(Environment< IValue > env) {
IValue v1 = left.eval(env);
if (v1 instanceof  VInt) {
   IValue v2 = right.eval(env)
   if (v2 instanceof VInt) {
        return new Vint((VInt)v1).getval()+((VInt)v2).getval())
}
throw InterpreterError("illegal types to + operator");
}
```

Imperative Language

EE ->

| **num** | **true** | **false** | **id**
| EE **+** EE | EE **-** EE
| EE ***** EE | EE **/** EE | **-**EE | **(** EE **)**
| EE **==** EE | EE **>** EE | EE **>=** EE | EE **~=** EE |**…**
| EE **&&** EE | EE **||** EE | **~** EE
| **def** (**id** = EE)+ **in** EE **end**
| **new** EE | EE **:=** EE | **!** EE
| **if** EE **then** EE **else** EE **end**
| **while** EE **do** EE **end**
| **print** EE | EE **;** EE

Imperative Language (abstract syntax)

EE ->

| num | true | false | id

| EE + EE | EE - EE

| EE * EE | EE / EE | -EE | (EE)

| EE == EE | EE > EE | EE >= EE | …

| EE && EE | EE || EE | ~ EE

| def (id = EE)+ in EE end

| new EE | EE := EE | ! EE

| if EE then EE else EE end

| while EE do EE end

| print EE | EE ; EE

new constructs

Parsing (concrete syntax)

You define it !

Parsing (concrete syntax)

```
S -> SE (; SE)*
SE -> BA (:= BA)*
BA -> BM (|| BM)*      // boolean additives
BM -> R (&& R)*        // boolean multiplicatives
R -> E ( > E )?        // relops
E -> T ( + T)*         // integer additives
T -> F ( * F) *        // integer multiplicatives
```

IValues (schematic)

```
interface IValue {
void show();
}

//Value constructors
VInt(n)
VBool(t)
VCell(value)
```

IValues (schematic)

```
class VInt implements IValue {
int v;
VInt(int v0) { v = v0; }
int getval() { return v;}
}
```

IValues (schematic)

```
class VCell implements IValue {
IValue v;
VCell(IValue v0) { v = v0; }
IValue get() { return v;}
void set(IValue v0) { v = v0;}
}
```

IValues (schematic)

```
class ASTAdd implements ASTNode {

IValue eval(Environmnent env) {
v1 = left.eval(env);
if (v1 instanceof  VInt) {
   v2 = right.eval(env)
   if (v2 instanceof VInt) {
        return new Vint((VInt)v1).getval()+((VInt)v2).getval())
   }
throw TypeError("illegal arguments to + operator");
}
```

IValues (schematic)

```
class ASTNew implements ASTNode {
IValue arg;
IValue eval(Environmnent env) {
IValue v1 = arg.eval(env);
return new VCell(v1);
}
```

IValues (schematic)

```
class ASTAssign implements ASTNode {
…
IValue eval(Environmnent env) {
IValue v1 = left.eval(env);
if (v1 instanceof  VCell) {
            v2 = right.eval(env);
            ((VCell)v1).set(v2);
      return v2;
throw TypeError(“illegal arguments to := operator”);

}
```

IValues (schematic)

```
class ASTPrintln implements ASTNode {
…
IValue eval(Environmnent env) {
        IValue v1 = arg.eval(env);
        System.out.println(v1.show());
}
```

Examples (more on lecture slides)

```
def a = new 5 in a := !a + 1; println !a end;;

def x = new 10
    s = new 0  in
while !x>0 do
    s := !s + !x ; x := !x – 1
end; println !s;
end;;
```

Typed Language

Imperative Language

EE ->

| **num** | **true** | **false** | **id**
| EE **+** EE | EE **-** EE
| EE ***** EE | EE **/** EE | **-**EE | **(** EE **)**
| EE **==** EE | EE **>** EE | EE **>=** EE | EE **~=** EE |**…**
| EE **&&** EE | EE **||** EE | **~** EE
| **def** (**id** = EE)+ **in** EE **end**
| **new** EE | EE **:=** EE | **!** EE
| **if** EE **then** EE **else** EE **end**
| **while** EE **do** EE **end**
| **print** EE | EE **;** EE

Goal

Implement a complete static type checker for the basic imperative-functional language specified

Use the approach developed in the lectures

- extend parser to support type declarations
- AST model for type syntax
- IType model for type values
- Environment based typechecker
- Integrate with your interpreter, before running the program, typecheck it!

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

Type (generic)

```
interface IType { /* represents types */
String show();
}

IType typecheck(Environment<IType> env) { ... }

// implementing classes
TypeInt()
TypeBool()
TypeRef(IType t)
```

Typechecker (schematic)

```
class ASTAdd implements ASTNode {

IType typecheck(Environment<IType> env) {
IType v1 = left.typecheck(env);
if (v1 instanceof  IntType) {
   IType v2 = right.typecheck(env);
   if (v2 instanceof IntType) {
        return v1;
   }
throw TypeError(“illegal arguments to + operator”);
}
```

Typechecker (schematic)

```
class ASTEqual implements ASTNode {

IType typecheck(Environment<IType> env) {
IType v1 = left.typecheck(env);
if (v1 instanceof  IntType) {
   IType v2 = right.typecheck(env);
   if (v2 instanceof IntType) {
        return boolType;
   }
throw TypeError(“illegal arguments to + operator”);
}
```

Typechecker (schematic)

```
class ASTDeref implements ASTNode {


IType typecheck(Environment<IType> env) {
IType v1 = exp.typecheck(env);
if (v1 instanceof  RefType) {
  Type reftype = ((RefType)v1).getType();
   nodetype = reftype;
   return reftype;
} else throw
     new TypeError("illegal argument to ! operator");
}
```

Compiler for Imperative Language

Goal

Implement a compiler for the imperative-language

Use the approach developed in the lectures

- Define a compile method in interface ASTNode to transverse the AST and generate code
- Use type information (from the typechecker) as needed to generate proper code
- code generation for the JVM (assemble with Jasmin)

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

Imperative Language

EE ->

| num | true | false | id

| EE + EE | EE - EE

| EE * EE | EE / EE | -EE | (EE)

| EE == EE | EE > EE | EE >= EE | EE ~= EE |…

| EE && EE | EE || EE | ~ EE

| def (id = EE)+ in EE end

| new EE | EE := EE | ! EE

| if EE then EE else EE end

| while EE do EE end

| print EE | EE ; EE

Phases

Use the approach developed in the lectures

- Implement the compilation method for boolean values, relational and boolean operations
- Implement the compilation method for reference operations
- Use type information (from the **typechecker**) as needed to generate proper code
- For that it may be useful to annotate identifiers and names in declarations in the AST with their types
- You **need type information** to define classes for frames and references, etc….

Final Handout

Interpretation and Compilation DEC-2021

Luis Caires

Goal

Implement interpreter and compiler for our imperative-functional language

Use the approach developed in the lectures

- Define a compile method in interface ASTNode to transverse the AST and generate code
- Use type information (from the typechecker) as needed to generate proper code
- code generation for the JVM (assemble with Jasmin)

Fully understanding the handout statement is part of the handout as well. Contact me if you need help.

Languages

Level 1 = basic imperative language

Level 2 = Level 1 + (recursive) functions

Level 3 = Level 2 + extension with record types

The 3 languages are described in the next slides

Level 1

EM -> E0(<;>E0)*	ASTSeq(E0,E0)
E0 -> E (<:=>E)?	ASTAssign(E,E)
E -> EA(< == > EA)?	ASTEq(EA,EA)
EA -> T(<+>EA)*	ASTAdd(E1,E2)
T -> F ((<*>T)*	ASTMul(F,T)

AL -> (EM(<,>EM)*)?

PL -> (id:Type(<,>id:Type)*)?

F -> **num** | **id** | **bool** | **def** (**id** = EM)+ **in** EM **end**	
| **new** F | <!> F	
| <(> EM <)>	
| **if EM then EM (else EM)? end**	ASTIf(EM,EM,EM)
| **while EM do EM end**	ASTWhile(EM,EM)
| **print** E	ASTPrint(E)
| **println** E	ASTPrintLn()

Level 2

EM -> E0(<;>E0)*	ASTSeq(E0,E0)
E0 -> E (<:=>E)?	ASTAssign(E,E)
E -> EA(< == > EA)?	ASTEq(EA,EA)
EA -> T(<+>EA)*	ASTAdd(E1,E2)
T -> F ((<*>T)*	ASTMul(F,T)
| (<(>AL<)>)*	ASTApply(F,AL)

AL -> (EM(<,>EM)*)?

PL -> (id:Type(<,>id:Type)*)?

F -> **num** | **id** | **bool** | **def** (**id (:** Type**)?**= EM)+ **in** EM **end**

| **new** F | <!> F	
| **fun** PL -> EM **end** | <(> EM <)>	
| if EM then EM (else EM)? end	ASTIf(EM,EM,EM)
| while EM do EM end	ASTWhile(EM,EM)
| **print** E	ASTPrint(E)
| **println** E	ASTPrintLn()

Type -> **int**| **bool** | **ref** Type | <(> Type (**<,>** Type)*) <)> Type

Level 3

Level 3 language introduces a data type of records and a data type of strings

The syntax for record expressions is

[id = EM ; id = EM ; id = EM ; …]

// record construction

F.id

// record field selection

Level 3 - Example

Example

```
def
   person1 = [ name = “joe”; age = 22 ]
   person2 = [ name = “mary”; age = 5]
in
  println  person1.age + person2.age
end
```

NOTE: this program prints out the value 27

Levels of Acomplishment

0 – Interpreter for Level 1 language

worth 14/20 points in final handout grading

1 – Interpreter for Level 2 language

worth 16/20 points in final handout grading

2 – Interpreter and Compiler for Level 1 language

worth 18/20 points in final handout grading

3 – Interpreter and Compiler for Level 2 language

worth 20/20 points in final handout grading

4 – Interpreter and Compiler Level 3 language

worth 22/20 points in final handout grading

submission deadline for final handout: January 15th 2022